Exploiting Memory Hierarchy for a Computational Fluid Dynamics Accelerator on FPGAs

Hirokazu Morishita† Yasunori Osana‡ Naoyuki Fujita†† Hideharu Amano†

† Department of Computer Science, Keio University, Yokohama, 223-8522, Japan
‡ Department of Computer and Information Science, Seikei University, Tokyo, 180-8633, Japan
†† Aerospace Research and Development Directorate
Japan Aerospace Exploration Agency, Tokyo, 182-8522, Japan
E-mail : cfd@am.ics.keio.ac.jp

Abstract

Computational Fluid Dynamics (CFD) is an important tool for aeronautical engineers. Instead of expensive supercomputers or clusters, using custom pipelines built on FPGAs is expected to be a cost effective solution to accelerate CFD. The problem is that to keep the pipeline busy is difficult because of the memory bandwidth. To deal with this problem, an effective memory access method using BlockRAMs is implemented based on a careful survey about memory access pattern. This work is targetting on two major subroutines in UPACS, a CFD software package. As a result, the amount of data transfer is reduced about 40%. This shows 46-170fold speed-up is expected by several Virtex-4 FPGAs compared to Itanium2 processor.

1. Introduction

Computational Fluid Dynamics (CFD) is a numerical method to analyze the behavior of the fluid. It has been known as an application that has massive floating point operations, and executed on large scale clusters or supercomputers. Recently, CFD is adopted tools for designing wings, jet engine and other aircraft components. Since wind tunnel experiments are costly parts in designing such aircraft components, it is becoming more important. A high precision CFD package is demanded by aeronautical engineers as well as aerodynamics researchers since there are various requirements for simulations.

UPACS (Unified Platform for Aerospace Computational Simulation [9]) developed by JAXA (Japan Aerospace Exploration Agency) is one of such CFD tools. Although it is a convenient tool for aerodynamics analysis, it often takes several days or weeks to solve large grids [16]. This is mainly caused by its low parallelism accompanied with pointer links and complicated memory access patterns, and this leads a performance degradation on massively parallel systems.

Reconfigurable systems using FPGAs have been utilized for acceleration of various applications including informatics, digital image processing and others. The early reconfigurable systems did not focus on large scale numerical scientific applications. However, the use of FPGAs for such areas has been growing by rapid performance improvements of the modern FPGA technology. BEE2 [1] and PROGRAPE [13] are good examples of successful acceleration for scientific applications using multiple FPGAs. Various supercomputers with reconfigurable units are available for reconfigurable supercomputing [10].

Our study aims to improve the performance of the UPACS by using a reconfigurable system with multiple FPGAs. As the first step, two subroutines are selected by the results of profiles. Then the arithmetic pipelines for these subroutines are designed [8]. The target problems employ sparse matrix computations with double precision elements, and the memory bandwidth limits the throughput. To make matters worse, there are dependencies between each input data, and it makes the utilization ratios of the arithmetic pipelines low. To deal with this problem, we designed a data controller to supply the arithmetic pipelines with data.

2. The UPACS

2.1. Acceleration of UPACS

UPACS is a CFD package to simulate a compressible flow using multi-block grids. Its purpose is to provide researchers with an easy way to run large scale simulations. Source code is written in Fortran 90 and supports the MPI interface. By choosing solvers, users can run simulations on their parallel systems without any code tunings. However, the performance of UPACS does not linearly scale to the number of nodes. This is caused by accesses of pointer lists for structures like `A[B[i]]` which compiler cannot optimize statically. Moreover, memory access patterns of

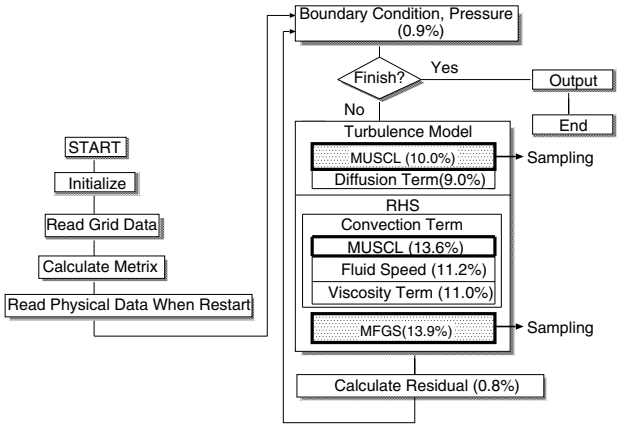

Figure 1. The Profiling Result of UPACS

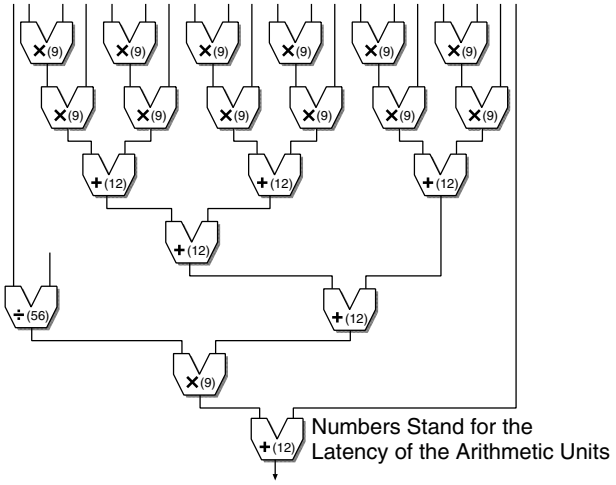

Figure 2. The Structure of Pipeline for MFGS

UPACS are too complicated. By such reasons, UPACS is not suitable to be executed on highly parallel systems [16]. In spite of these problems, many users need for the speed of UPACS execution.

To accelerate UPACS, Cell/BE, GPGPUs, and the other accelerators can be the next candidates of a high speed execution platform. A hardware implementation with memory modules and dedicated accelerator like GRAPE-DR [5] is also a hopeful approach from the viewpoint of performance.

In this work, FPGAs are chosen to exploit complicated fine-grain parallelisms by pipelines and memory controllers.

Implementing CFD on FPGAs was difficult in the previous decade because of its heavy usage of floating point operations and data. However, along with the rapid progress of FPGAs, challenges to develop such systems had started[7] [6][4][11][14].

Although the previous approaches are practical, the target is not a software package to be applied on variety problems. Our target is a CFD package to be used in designing air craft components. In order to implement the target application on FPGA, problems including double precision computations [2] with complex memory accesses must be coped with.

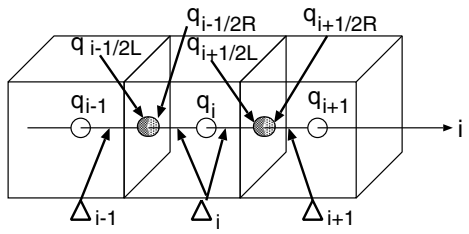

Figure 3. The Algorithm of MUSCL Method

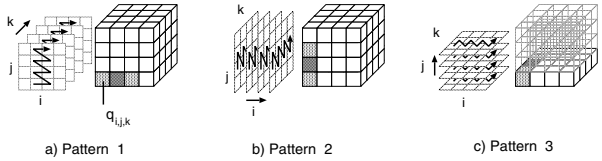

Figure 4. Memory Access Pattern of MUSCL

2.2. Profiling

As the early stage of this study, we profiled execution time of UPACS on SPARC64V processors at 1.3GHz with Solaris8 OS. As shown in Figure 1, the execution time of UPACS is mainly consumed in the main loop distributed into several procedures. Amdahl's law indicates that subroutines with large computation time should be selected. On the other hand, subroutines with a complicated data dependency are hard to be accelerated. Considering them, two subroutines are selected as our first step: MFGS (Matrix-Free Gauss-Seidel) and MUSCL (Monotone Upstream-centered Schemes for Conservation Law) in the turbulence model. MFGS has the largest portion in execution time (13.9%), however, it has a complicated data dependency. MUSCL in the turbulence model has a relatively simple data dependency while consuming 10.0% of total execution time. In addition to these two subroutines (23.9%), MUSCL is also used in the convection term. Although this one is slightly different from another in the turbulence model, it is easy to modify for the turbulence model so that our coverage will be almost 37.5%. In our preliminarily implementation [8], the data flow graphs of the subroutines are mapped directly onto FPGA. Figure 2 shows the arithmetic pipeline for MFGS subroutine.

The arithmetic pipeline has an excellent potential throughput, however, the memory bandwidth requirement of this design was not realistic. This shows that the mem-

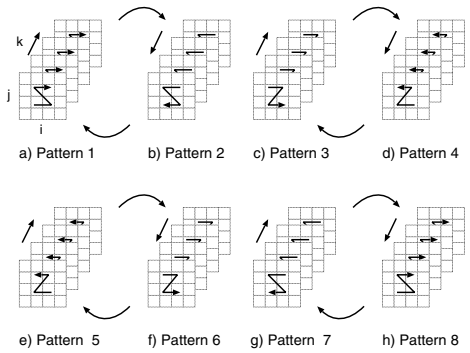

Figure 5. Memory Access Pattern of MFGS

ory accesses are the key for acceleration [12]. One of the large motivation of this work is to propose a methodology to use the arithmetic pipelines efficiently.

3. Algorithm analysis

3.1. Target Algorithms

MUSCL extrapolates contact surface values from cell center values shown in equations (1) to (4),

$$q'_{I+1/2} = (q_{i+1} - q_i)/(\Delta_{i+1} + \Delta_i), \quad (1)$$

$$q'_{I-1/2} = (q_i - q_{i-1})/(\Delta_i + \Delta_{i-1}), \quad (2)$$

$$q_{i\pm1/2} \cong q_i \pm \psi(r)\Delta_i q'_{I-1/2}, \quad (3)$$

$$r = q'_{I+1/2}/q'_{I-1/2}, \quad (4)$$

where q_i is the cell center value, Δ_i is the distance between the cell center and the contact surface, and $q_{i\pm1/2}$ is the contact surface value as shown in Figure 3. i in the equations (1) to (4) is the direction which can extend to three dimensions. Although UPACS supports various limiter functions, Van Albada limiter shown as the equation (5) was chosen.

$$\psi(r) = (r^2 + r)/(r^2 + 1) \quad (5)$$

MFGS method [3] performs time integration. MFGS requires the upper triangular matrix, lower triangular matrix and reversed diagonal matrix. This method is faster than the original Gauss-Seidel method since it approximates the reverse process and roughly cuts off the iteration.

Number of iteraions in MUSCL and MFGS are proportional to the grid size, and they can be processed in parallel on FPGAs. Fortunately, MUSCL has low data dependencies so that it is executable quickly unless the data transfer stops. MFGS has data dependencies which need a sophisticated mechanism to transfer data into the pipeline as possible.

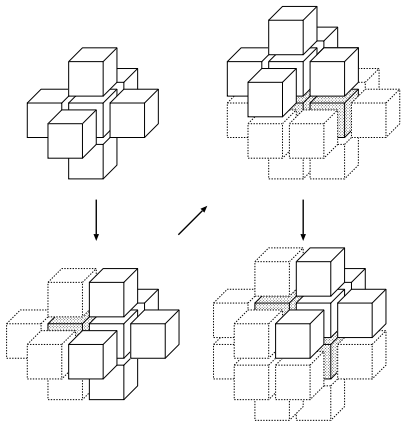

Figure 6. Data Dependency of MFGS

3.2. The Memory Access Pattern

In both subroutines, data is stored in three dimensional structure, and the computation is performed toward a certain direction of the structure. The continuous computation is called "sweeping", and the order of sweeping is different among the subroutines. MUSCL scans the 3D space continuously in three patterns as shown in Figure 4, and each step of sweeping needs three data: one is for the target point itself, and others are for two adjacent points selected by sweeping patterns. In pattern 1, MUSCL sweeps direction i first, j second, and k at last. Then in pattern 2, j is swept first, then k and i swept. In pattern 3, the order becomes the k, i and j. The sweep pattern is given by the host system, and it is not changed once calculation is started.

On the other hand, MFGS refers seven data points in the every iteration, including the point of interest and 6 adjacent points in all three dimensions. As shown in Figure 5, MFGS has total 8 sweep patterns since each 4 pattern has forward sweep and successive backward sweep. Only one pattern among the 8 patterns is described in this paper, however, the others are already implemented in the same manner. MFGS also has a strict data dependency. Figure 6 shows how the calculation is done in the pattern 1. Since the results are reused in successive calculations, the arithmetic pipeline stalls until the results are available.

Besides the data dependencies, the problems of the I/O pins of FPGAs must be considered. Table 1 shows the I/O pin requirements of MUSCL and MFGS. MUSCL needs 3×6 input ports and 10 output ports, and MFGS needs $7\times1+7$ input ports and 1 output port. Thus, these two methods require 28 ports and 15 ports respectively. As the number of pins are limited in both FPGAs and external memory devices, handling of I/O data must be considered.

Table 1. I/O Pin Requirements

	MUSCL	MFGS
Input		
Data With Adjacent Two Cells	6	0
Data With Adjacent Seven Cells	0	1
Data Without Neighbor Cells	0	7
Output		
Result	10	1
Total Numbers of Data	28	15

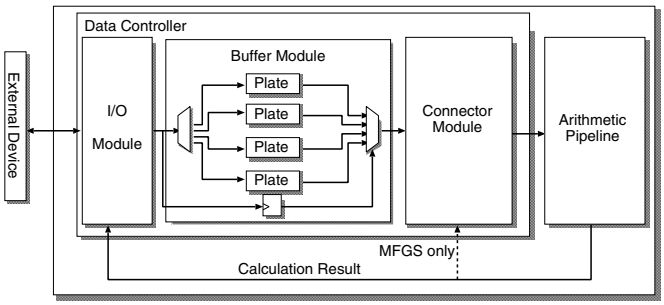

Figure 7. The System Overview

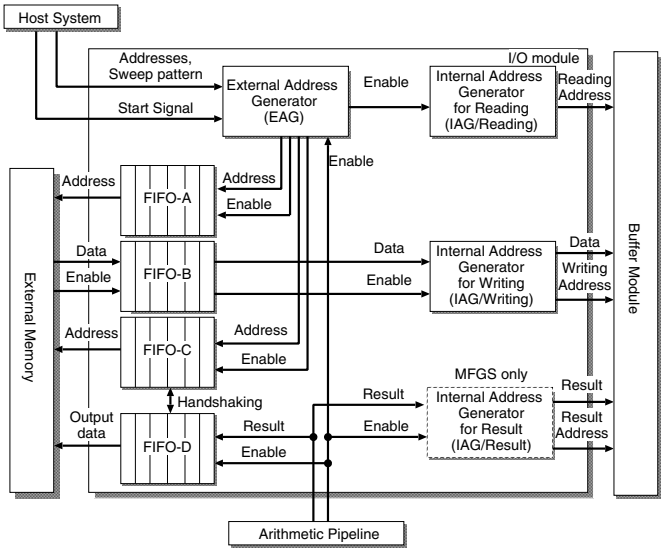

Figure 8. The Concept of I/O Module

4. Implementation

4.1. System Overview

For efficient 3 dimensional data access to the external memory, the data controller including several BlockRAMs as buffers has been designed. The organization of modules is shown in Figure 7. Both in MUSCL and MFGS, the data controller consists of the four modules.
(1) I/O module: handles following three data streams:
(A) from/to external memory, (B) to Buffer module, and (C) from Arithmetic pipeline.
(2) Buffer module: holds the partial data temporarily to supply to Arithmetic pipeline. It has four plate modules, each to keep a small 2 dimensional dataset on BlockRAM. By having multiple plate modules, latency of external memory is hidden.
(3) Connector module: solves the data dependency. Since the number of ports for BlockRAM is limited, Connector module optimizes the timings of reading operations.
(4) Arithmetic pipeline: structured by arranging the computational units in the form of the data flow.
The major differences between MUSCL and MFGS is the structure of Connector module, data path to the Connector module, and the number of address generators in I/O module. These differences are caused by the difference of data dependency between both algorithms. The detail of these modules are described in the following section.

4.2. I/O Module

The role of I/O module is to adjust the timing of the calculation by sending addresses to Buffer module so that the access latency is hidden. The structure of I/O module is shown in Figure 8. FIFOs are used as buffer storage between external memory and internal modules. EAG (External Address Generator) sends addresses for loading data and storing computation results from/to the external memory with enable signals to IAG/Reading (Internal Address Generator for Reading). EAG in MUSCL sends address adapted with the patterns to sweep uniformly in IAGs. MFGS sweeps as shown in Figure 9-b) to fill Arithmetic pipeline. IAGs send addresses and data to scan array in Buffer module with the signal indicating the calculation point. The structure of the address generator in MUSCL and MFGS are different because of their memory access patterns, and MFGS has an IAG/Result (Internal Address Generator for Result), since the data must be restored into BlockRAMs.

All the address generators consist of the state machine with counters, and send addresses to Buffer module. They also send signals to control the order of computation in Connector module. I/O module works as follows.
(1) Receives a signal to start calculation, the pattern of sweeping, and the start/end addresses of the calculation points from the host system.
(2) Calculates an external address and sends it to FIFO-A and FIFO-C with enable signals. The address is calculated in accordance with sweep patterns.
(3) When IAG/Reading receives the enable signal, it calculates the address for Buffer module. This address is used as reading the BlockRAM in Buffer module.
(4) While receiving data from the external memory, FIFO-B

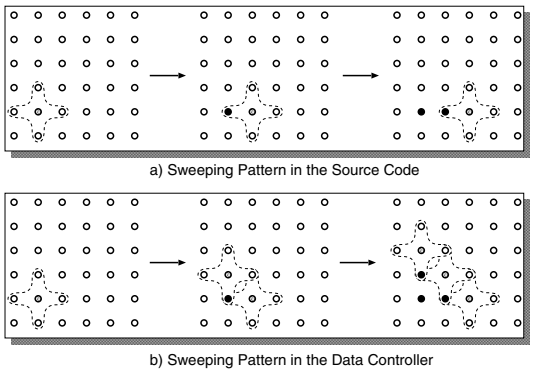

Figure 9. Sweeping Methodology in MFGS

sends data with the enable signal to IAG/Writing.
(5) When IAG/Writing receives the enable signal and data, it sends both address and data to Buffer module. These are used as writing the BlockRAM in Buffer module.
(6) When I/O module receives result from Arithmetic pipeline, it writes data into the FIFO-D. In addition, when IAG/Result in MFGS receives result and enable signal, it sends result and address to Buffer module. These are also used for writing the BlockRAM in Buffer module.
(7) Iterate from (2) to (6).

4.3. Buffer Module

The role of Buffer module is to keep Arithmetic pipeline busy by preloading data on BlockRAM, and reducing the amount of data transfers from external memory. As shown in Figure 7 and also described in Section 4.1, the plate modules consist of BlockRAMs to store data required in Arithmetic pipeline. Each of the plate module covers the k direction shown in Figure 4 and 5. 42×40 data points in MUSCL and 42×42 in MFGS are stored in BlockRAMs with 64 bit $\times$ 2048 words. This size is corresponding to the test case described in Table 5. These are larger than the problem size (40×40 in Table 5) because plate modules have to keep boundary areas in addition. Although the number of the plate modules should be increased as FPGA resources allow, four plate modules are adequate to run. Buffer module works as follows.
(1) The data are stored with the consideration of sweep patterns before calculation starts.
(2) When it receives an address or data from IAG/Reading or IAG/Writing in I/O module, it sends them to an appropriate plate module.
(3) When data comes from the plate module, it sends the data to Connector module.

4.4. Connector Module

Connector module manages the dependency and interval of data transfer to Arithmetic pipeline. Since interleaving needs more sophisticated address generators, dual ports BlockRAMs are adopted to realize Data controller. In order to inject data to the Arithmetic pipeline continuously, Connector module in MUSCL has four registers: Left, Center, Right and Temporary register. It works as shown in Figure 10. Note that only two can be read out from the memory at once.

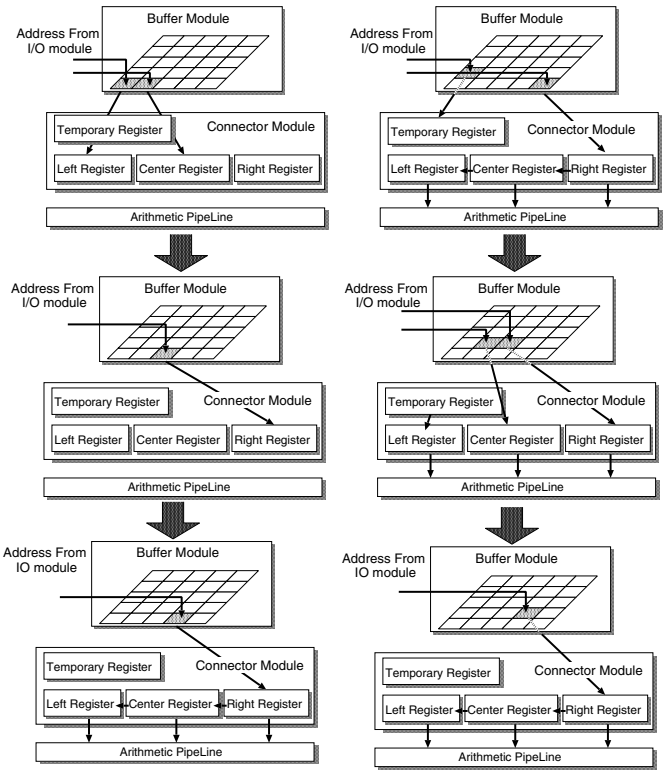

Figure 10. MUSCL Connector Module

(1) The first two points are accessed and stored in Left and Center register.
(2) The next point is read and stored to the Right register.
(3) The next point is taken from memory. At the same time, data in registers are shifted: Center to Left, and Right to Center. New data from memory is stored to Right register.
(4) When memory access reaches to the edge, the first data in the next row is stored in Temporary register.
(5) The next two points of the memory are read and stored in Center and Right register. In this step, registers are not shifted.
(6) Iterate from (3) to (5) until the calculation ends.
By this mechanism, data are sent from the registers to Arithmetic pipeline in every cycle expect for the first two steps.

Although Arithmetic pipeline can be filled with the connector module with four registers in MUSCL, more complicated mechanism is required in MFGS to cope with complex access pattern (Figure 9 and 11). If the memory is simply swept to the row or column direction, we cannot pre-

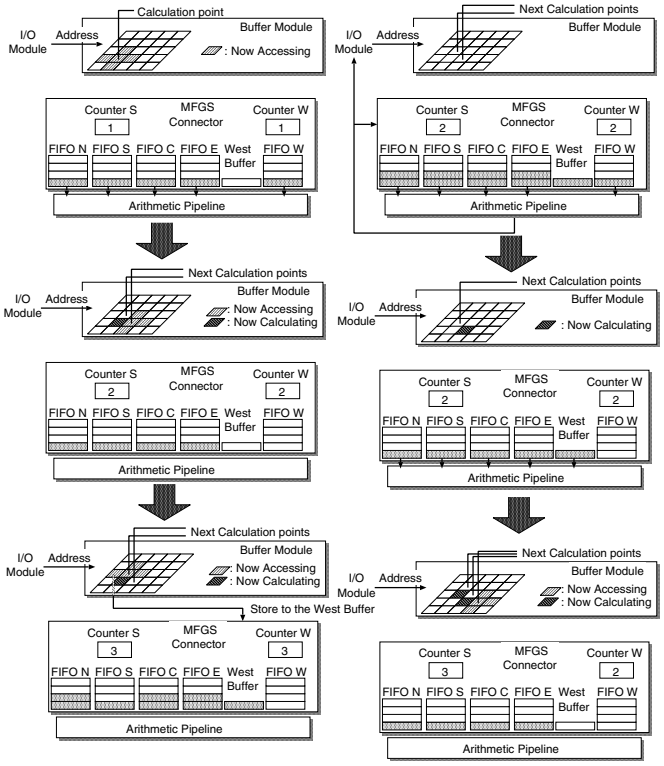

Figure 11. MFGS Connector Module

pare enough data to send to Arithmetic pipeline. In order to solve this problem, the sweeping process is modified to be done in diagonal direction to Buffer module. To synchronize variables for the seven data points, seven FIFOs are employed (Figure 11). There are FIFO-N, FIFO-S, FIFO-C, FIFO-E and FIFO-W. For example, FIFO-N is for northern neighbor of the calculation point. In addition to five FIFOs in Figure 11, there is also FIFO-U for upper neighbor and FIFO-L for lower neighbor. Variables are transferred when
i) All FIFOs have valid data.
ii) All FIFOs but FIFO-W have only 1 valid data.
FIFO-W is empty, and the west buffer has a valid data.
FIFO-S and FIFO-W have a counter to maintain dependencies. The counters are incremented when corresponding FIFOs have to wait for a result from Arithmetic pipeline, and decremented when values of the counters are referred to determine the FIFOs to inject a result from Arithmetic pipeline. The west buffer prefetches western point while FIFO-W is waiting for a result from Arithmetic pipeline. Following is the rule of Connector module's behavior.
(1) The first point and its neighbors are read in three clock cycles, and stored to corresponding FIFOs. Counters are incremented since this result will be required in following calculation points. As the variables become available, data is sent to Arithmetic pipeline.
(2) The next data point and its neighbors are fetched and stored in FIFOs. Fetching the west boundary is an exception. In this case, the value is stored in the west buffer instead of FIFO-W.
(3) When a result comes out from Arithmetic pipeline, the value is sent to I/O module and FIFOs where the counters indicate. This satisfies condition i) or ii) above to send data to Arithmetic pipeline.
(4) Iterate from (2) to (3) until the calculation ends.
This procedure increases pipeline utilization ratio gradually, while maintaining the dependency properly.

Table 2. Assumed FPGA Resource

	Slice	DSP48	BlockRAM
XC4VLX200	89,088	96	336
XC4VLX100	49,152	96	240
XC4VSX55	24,576	512	320
XC4VSX35	15,360	192	192

Table 3. Resource Requirement of MUSCL

	Slice	DSP48	BlockRAM
Arithmetic Pipeline	149,175	712	0
Buffer Module	10,299	0	192
I/O Module	1,057	0	38
Total	160,531	712	230

Table 4. Resource Requirement of MFGS

	Slice	DSP48	BlockRAM
Arithmetic Pipeline	20,467	18	0
Buffer Module	2,722	0	256
Connector Module	881	0	28
I/O Module	1,337	0	24
Total	25,407	18	308

Table 5. Environment for UPACS Execution

Machine	Fujitsu PRIMERGY RXI300
CPU	Itanium2 1.5GHz
OS	Red Hat Enterprise Linux AS release 3
Compiler	Intel Fortran V9.0
Grid Size	$40 \times 40 \times 40$
Iteration	100 times

4.5. Arithmetic Pipeline

Arithmetic pipeline is designed by arranging Arithmetic units. Double-precision functional units are made by Xilinx Core Generator (floating point 3.0). MUSCL has more floating point units than MFGS since it deals with five physical values in parallel. Although its resource is easily reduced, it must be designed so as not to be a bottleneck of Arithmetic pipeline.

Table 6. Pin Requirements of the System

	MUSCL	MFGS
Input		
Data With Adjacent Two Cells	0	0
Data With Adjacent Seven Cells	0	0
Data Without Neighbor Cells	6	8
Output		
Result	10	1
Total Numbers of Data	16	9

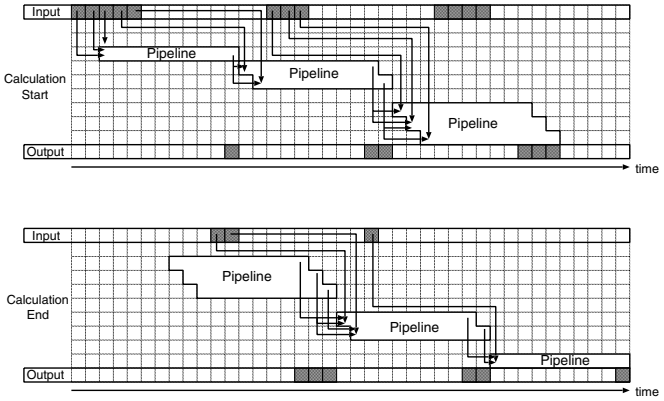

Figure 12. Conception of Arithmetic Pipeline Scheduling of MFGS

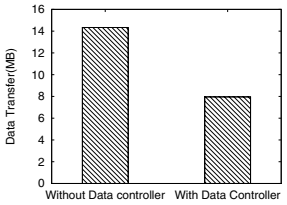

Figure 13. Data Transfer of MUSCL

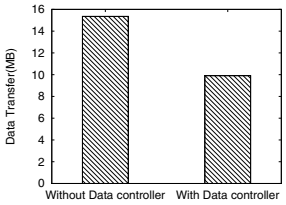

Figure 14. Data Transfer of MFGS

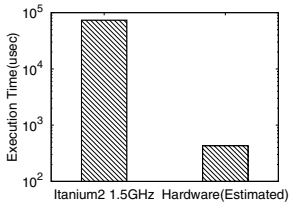

Figure 15. Execution Time of MUSCL

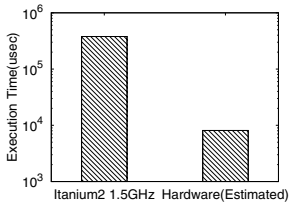

Figure 16. Execution Time of MFGS

5. Evaluation

MUSCL and MFGS are designed with Verilog-HDL for description and ISE 8.2.03i for synthesis, and evaluated the required resource, performance and error ratio.

5.1. Resource Evaluation

The amount of slices for random gates, DSP48 used as a part of floating point units, and BlockRAM for Buffer modules are evaluated. Xilinx Virtex-4 FPGAs [15] shown in Table 2 are selected as target devices. The resource for MUSCL and MFGS are shown in Table 3 and Table 4, respectively. MUSCL does not have Connector module, and uses more slices for buffer module compared with MFGS. This is because the buffer module in MUSCL contains address generators and Connector module.

As a result, MUSCL requires two XC4VLX200s, an XC4VSX55 and an XC4VSX35 to be implemented. For MFGS, two XC4VLX100s are sufficient.

5.2. Performance Evaluation

Since the reduction of the data transfer is a key to improve the performance, the effect of the data controller is evaluated. Data are assumed to be sent to Arithmetic pipeline directly from the external memory although the limit of the pin cannot meet. Data transfers of MUSCL and MFGS are calculated by equation (6) and (7) respectively.

$$P \times G \times 8 \quad (\text{Bytes}) \tag{6}$$

$$P \times G \times 8 \times 2 \quad (\text{Bytes}) \tag{7}$$

P is the number of ports, G is the number of grid size, and MFGS needs a backwards sweep so that the result needs to be doubled. According to Table. 1 and Table 5, MUSCL pipeline transfers $18 \times 40^3 \times 8 = 9.22$MB for input, and $10 \times 40^3 \times 8 = 5.12$MB for output. Similar to MUSCL, MFGS pipeline transfers $14 \times 40^3 \times 8 \times 2 = 14.34$MB for input and $1 \times 40^3 \times 8 \times 2 = 1.02$MB for output.

On the other hand, data controller suppresses all redundant transfers for referring adjacent points as shown in Table 6. In addition, a part of data are stored in three plate modules beforehand. As a result, MUSCL reads $6\times (42 \times 40\times(40-3)) \times 8 = 2.98$MB from external memory, and MFGS reads $8\times (42\times 42\times(42-3)) \times 8 \times 2 = 8.81$MB data transfers.

In this system, both MUSCL and MFGS transfer as same data as Arithmetic pipeline only for output. Data transfers for input in MUSCL and MFGS are reduced by about 67.7% and 38.6% respectively and Figure 13, 14 shows total reductions. This results indicate that 44.5% and 35.5% data transfers in MUSCL and MFGS are reduced.

Secondly, performance is measured by comparing software execution with the hardware implementation. Since the target system must be distributed into several chips, the

evaluation must include the overhead between inter-chip communication. Estimated overhead is about 15 clock cycles because of converting parallel/serial data in order to accommodate a serial link of the target devices. In addition, the overhead of the communication between software and hardware using PCI-EX must be considered. However, in order to calculate the potensial of this work, the hardware performance without overhead of those communication is estimated by RTL simulation. The environment of the software execution is shown in Table 5.

To begin with, we evaluated how many clocks modules require to finish calculation. The reductions of data transfers affects the clock cycles because it increases the utilization of Arithmetic pipeline. Assuming that the data are successively input to Arithmetic pipeline in MUSCL, it takes $380\ (\mathrm{clocks}) + (40 \times 40 \times 40)\ (\mathrm{points}) = 64380$ clocks.

However data for Arithmetic pipeline in MFGS cannot be prepared in every clock cycle, sweeping is done as shown in Figure 12. The appearance of the results from Arithmetic pipeline is observed, and it took about 80000 clocks from the RTL simulation.

Then the critical path of the designed system is analyzed and the maximum frequency is calculated to predict the execution time on FPGA. As a result, MUSCL and MFGS can work at about 150MHz and 98.3MHz, respectively. The execution time comparison results are shown in Figure 15 and Figure 16. From them, we can estimate that MUSCL is about 170 times faster, while MFGS is about 46.8 times faster than Itanium2 1.5GHz processor.

5.3. Computational Error

The computational error ratio is evaluated by comparing the software execution with the hardware execution of MUSCL. As a result, 1.6×10^{-12} for maximum error ratio corresponding to the errors of lower two bits was estimated. This result shows that 15 digits out of the 17 digits in decimal were accordant. This accuracy is adequate, considering 52 mantissa bits of double precision corresponding to approximately 16 digits in decimal.

6. Conclusion

A part of software tool for CFD was implemented on FPGAs. Tuning memory access with Data controller to keep Arithmetic pipeline usage high, the performance of MUSCL and MFGS will be about 170 times and 46.8 times faster than software execution.

Currently, only 23.9% of total execution time is accelarated, and the accelation of the remaining part is our future work. In addition, memory access patterns continue to be analysed in order to detect the generality which is useful for an automation. The prototype board which includes an FPGA, two DIMM slots and four high speed serial links is now under development by JAXA, and the implementation using multiple boards is our next challenge.

References

[1] Chen Chang, John Wawrzynek, Robert W. Brodersen. BEE2:A High-End Reconfigurable Computing System. In *Configurable Computing:Fabrics and Systems*, pages 114–125, 2005.

[2] E. El-Araby, I.Gonzalez, T.El-Ghazawi. Beinging High-Performance reconfigurable Computing to Exact Computations. In *International Conference on Field Programmable Logic and Applications 2007*, 2007.

[3] E.Shima, A.Ochi, T.Nakamura, S.Saito, T.Iwamiya. Unstructured Grid CFD on Numerical Wind Tunnel. In *Parallel Computational Fluid Dynamics:Development and Applications of Parallel Technology*, 1998.

[4] Esther Andres, Carlos Carreras, Gabriel Caffarena, Maria del Carmen Molina, Ocrabio Nieto-Taladriz, Francisco Palacios. A Methodology for CFD Acceleration through Reconfigurable Hardware. In *AIAA Aerospace Sciences Meeting and Exhibit*, 2008.

[5] Junichiro Makino, Kei Hiraki, Mary Inaba. GRAPE-DR: 2-Pflops massively-parallel computer with 512-core, 512-Gflops processor chips for scientific computing. In *Supercomputing*, 2007.

[6] Kentaro Sano, Oliver Pell, Wayne Luk and Satoru Yamamoto, Satoru Yamamoto. FPGA-based Streaming Computation for Lattice Bolzman Method. In *Internaltonal Conference on Field Programmable Technologies*, pages 233–236, 2007.

[7] Kentaro Sano, Takanori Iizuka, Satoru Yamamoto. Systolic Architecture for Computational Fluid Dynamics on FPGAs. In *Field-Programmable Custom Computing Machines*, pages 107–116, 2007.

[8] Naoyuki Fujita, Takashi Nakamura, Yuichi Matsuo, Katsumi Yazawa, Yasuyuki Shiromizu, Hiroshi Okubo. Feasibility Study of CFD Code Acceleration using FPGA. In *Supercomputing*, 2007.

[9] Takaki Ryoji, Kazuomi Yamamoto, Takashi Yamane, Shunji Enomoto, Junichi Mukai. The Development of the UPACS CFD Environment. In *High Performance Computing, Proceedings of ISHPC*, pages 307–319, 2003.

[10] T.El-Ghazawi. High-Performance Reconfigurable Computing. In *Tutorial on International Conference on Field Programmable Technology*, 2007.

[11] Thomas Hauser. A Flow Solver for a Reconfigurable FPGA-Based Hypercomputer. In *AIAA Aerospace Sciences Meeting and Exhibit*, 2005.

[12] Tomoyoshi Kobori, Tsutomu Maruyama. A High Speed Computation System for 3D FCFC Lattice Gas Model with FPGA. In *International Conference on Field Programmable Logic and Applications*, pages 755–765, 2003.

[13] Tsuyoshi Hamada, Naohito Nakasato, Toshikazu Ebisuzaki. A 236 Gflops Astrophysical Simulation on a Reconfigurable System. In *Field-Programmable Custom Computing Machines*, 2005.

[14] William D.Smith, Austars R. Schnore. Towards an RCC-based accelerator for computational fluid dynamics applications. In *Supercomputing*, pages 239–261, 2004.

[15] Xilinx co. ltd. XC4000 series documents. 2007.

[16] Yuichi Matsuo, Masako Tsuchiya. Early Experience with Aerospace CFD at JAXA on the Fujitsu PRIMEPOWER HPC2500. In *Conference on High Performance Networking and Computing*, page 11, 2004.